Easy68k

A Beginner's Guide

Part I: Introduction

What is Easy68k?

Easy68k is a so-called 'programming environment' which many people apparently use in order to write programs in assembly language. According to the Easy68k website, (http://www.easy68k.com/) "EASy68K is a 68000 Structured Assembly Language IDE" which "allows you to edit, assemble, and run 68000 programs". For the time being, we will ignore the questionable usage of the words "allows" and "run", and accept this as our working definition of the Easy68K environment.

Assembly Language? What's that?

An assembly language is a low-level programming language, meaning it has relatively direct correspondence to the kinds of basic instructions that make your computer work. A good assembly language is made up of discrete instructions which can be translated into hexadecimal or binary with a minimum of effort. Using Easy68k as an emulator, you can write and assemble instructions in the Motorola 68K language with all the dexterity of a drunken rhinoceros carrying an infant across a tightrope.

If Easy68k is just an emulator, can I write 68k programs in another environment?

Yes, but according to Google, Easy68k is the #1 68k assembler and simulator, so you may as well just bite the bullet and take it like a man.

I've used Python/Java/C++/Ruby/etc., but I'm new to assembly languages. Can I make the same kinds of programs with Easy68k that I can with other languages/environments?

Well… kind of. Because assembly languages are so low-level, even the most basic of programming tasks—such as creating for loops, storing variables, displaying text, running if statements, performing arithmetic operations, and calling methods in 68k will usually require more code and always be less intuitive than in any other language. Furthermore, the Easy68k environment contains various bugs that will sabotage your code quietly and efficiently, like a thief in the night.

Part II: A Forensic Reconstruction of Easy68k's Development

At some point during the mid-80s, Easy68k was perpetrated by a mad computer scientist named Professor Charles Kelly, who was teaching at Monroe Community College. According to his website, he created the emulator by combining an editor one of his students had written for the Teesside assembler with source code for an incomplete 68k assembler and simulator. After some patching up, Professor Kelly decided it was time to release Frankenstein emulator upon the world that had rejected him with a cry of "Hello, cruel world".

During the mid-2000s, Professor Kelly appears to have been active on the Easy68k forums. Out of evident remorse for his creation, he gives computer science students advice on defeating Easy68k.

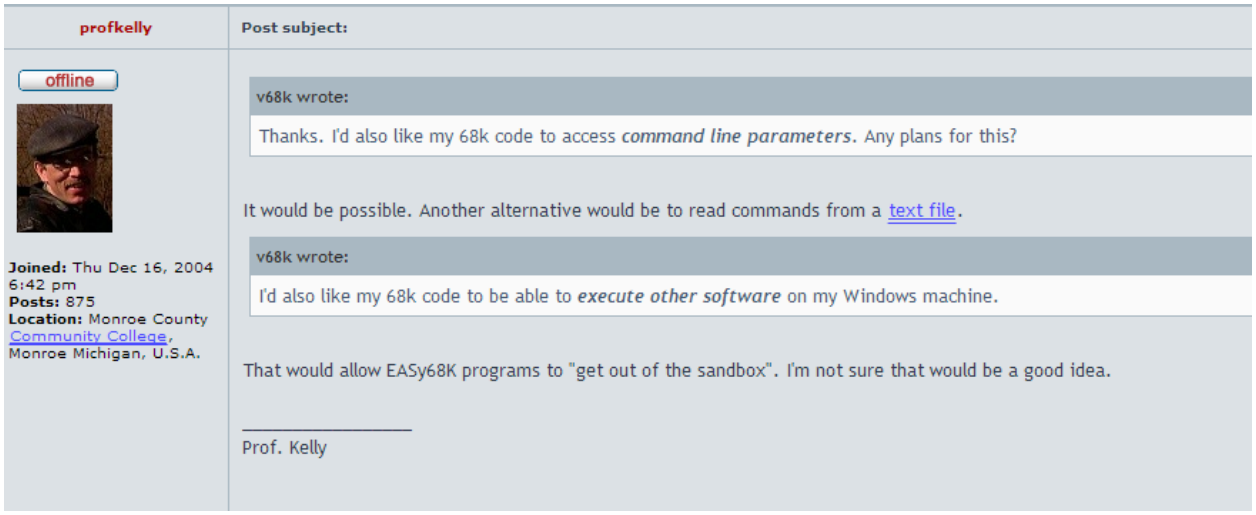


Fig 1: Kelly warns the foolish "v68k" not to give Easy68k access to the rest of his computer.

If at any point you decide to use Google for any information about Easy68k, be prepared to get an answer from the Professor himself. For example, Googling "Easy68k for loop" will yield a link to a thread on the Easy68k forum, in which Kelly kindly explains to "lostin68k" how to make a for loop. You can find very similar results by searching "Easy68k if statement", "Easy68k array", "Easy68k address error", "Easy68k program counter with displacement", "Easy68k movem", or "Easy68k is kill"—all of these and many, many more will show you the official Easy68k forum as the first or second result, with a reply from Professor Kelly as the first or second reply. Try it yourself! (Just don't expect to find any other documentation on how to use Easy68k, since hardly anyone uses it.)

Part III: Easy68k Basics

How can I make a hello world program in Easy68k?

Conveniently enough, the Easy68k website contains a tutorial which will teach you how to accomplish this very thing! Here is what the code looks like:

```
START   ORG     $1000

*-------Code for output------
        LEA     MESSAGE,A1      Loads MESSAGE into address register A1
        MOVE.B  #14,D0          Moves the number 14 into data register D0
        TRAP    #15             Displays Message

        MOVE.B  #9,D0
        TRAP    #15             Halt Simulator

CR      EQU     $0D             ASCII code for Carriage Return
LF      EQU     $0A             ASCII code for Line Feed
MESSAGE DC.B    'HELLO CRUEL WORLD',CR,LF
        DC.B    'ALL YOUR BASE ARE BELONG TO US',CR,LF
        DC.B    'YOU CAN ALSO MAKE MULTIPLE LINES',CR,LF
        DC.B    'IN ONE DC.B STATEMENT',CR,LF,'JUST REMEMBER',CR,LF
        DC.B    'TO SEPARATE EACH WITH A COMMA',CR,LF,CR,LF
        DC.B    'AND END YOUR LAST STATEMENT WITH A ZERO',CR,LF,0

        END     START
```

Fig. 2: Hello cruel world, indeed.

Notice that like a typewriter, Easy68k requires you to manually handle carriage return and line feed in order to go to the next line.

How can I store variables in Easy68k?

You can store your variables in data registers, address registers, or in specified memory locations in your program, like so:

```
MOVE.B          #10,D3      ; store the value 10 in data register 3.
MOVE.B          #100,$2000  ; store the value 100 at address $2000 in memory.
MOVE.W          A3,D4       ; copy whatever is in address register 3 to data register 4.
MOVE.W          D4,A5       ; causes an error because you must use MOVEA for An destinations.
MOVE.B          A3,D4       ; causes an error because fuck you, that's why.
```

However, this is often a bad practice because you can only have eight registers, and storing variables in memory may overwrite your program.

Instead, you generally use the DS pseudo-OPcode to reserve space in memory for your variables, and then

use MOVE commands to put data in these spaces. Like Python, Easy68k does not require strong typing of these variables. Unlike Python, Easy68k is utter shit, as you will learn after using it for five minutes.

How can I make classes in Easy68k?

Classes? Please excuse me while I laugh hysterically and masturbate out an open window.

Okay, I'm back. To answer your question, you cannot make classes in Easy68k. Instead, you must store all code for a project in one file, no matter how many thousands of lines of code you write. In place of using classes, you can simply use comments to separate regions of your code, close your eyes, and imagine what your program would be like with classes.

How can I make methods in Easy68k?

Most high-level programming languages allow you to write isolated, named methods with a set (or sets) of possible arguments (input), and then call these methods in other parts of the program, passing in whatever arguments are appropriate to the situation. 68k allows you to write subroutines, which are functionally similar to methods, but eschews the "declare possible argument types, call method with specific arguments" structure in favor of a "manually manage all of your local variables yourself literally every time you call any subroutine" structure.

How can I make a for loop in Easy68k?

The closest approximation to a for loop in 68k can be achieved through the use of a looping subroutine. For example, consider the following code, which you can use to raise 2 to the power 10:

```
RAISE_2_TO_10TH MOVE.B #10,D4      ; D4 says how many times you should loop.
                CLR    D3          ; D3 keeps track of your current index.
                MOVE.B #1,D2       ; D2 keeps track of your desire to kill yourself.
                BSR    RAISE_POWER
                ...
; for (int i = 0; i < D4; i++) (C++)
; for i in xrange(0,D4): (Python)
RAISE_POWER     MULU.W #2,D2
                ADDI.B #1,D3
                CMP    D4,D3
                BLT    RAISE_POWER
                RTS
```

If you want to make it clear what range of values your for loop actually traverses, you can simply follow the example above and add a comment imitating a C++ or Python-style for loop every time you make a for loop. That shouldn't be too tedious, right?

I am new to programming. Is Easy68k a good place for me to start?

No.

I am an experienced programmer. Is Easy68k a good tool for me to create programs quickly and efficiently?

No.

You should never use Easy68k under any circumstances.

Part IV: Memory Management and You

As an emulator, Easy68k thankfully cannot send its assembled instructions to your computer's memory, but instead simulates assembly-based memory manipulation when you execute your program. Upon execution, you should see this screen:

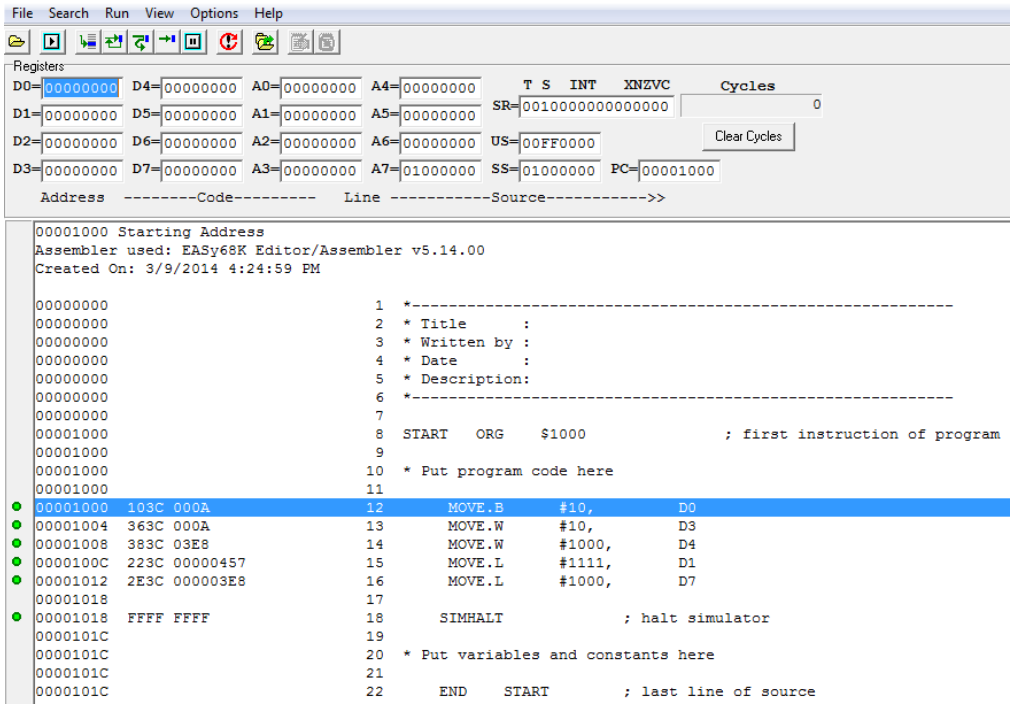


Fig. 3: No, there are not any jokes hidden in the code.

To briefly summarize what each component in the top pane means:

- D0-D7 are you data registers which you can use to store temporary values. D0, D1, and D2 have special applications, and you should be very careful about using them in or near parts of your code where you perform certain tasks, like printing output.
- A0-A7 are you address registers, all of which you can use as temporary variables. However, you shouldn't use A7 for this sort of thing, because Easy68k uses it as a reserved value, like SR, US, SS, and PC. However, there is no indication of this and you can still use A7 like any other address register, which can cause all kinds of errors.
- XNZVC are all bit values which are set by almost every instruction you can use in Easy68k, and with different meanings for all of them. This is usually done as a workaround so you can use certain kinds of comparison operations like "greater than" or "equal to" without actually altering any data in your memory.

- PC is the program counter, which indicates where the next instruction to be executed is located in your memory.
- "Cycles" refers to the number of clock cycles that your program has used so far.
- I have no idea what any of the other things mean and I don't care enough to look them up.

An important part of memory management is being aware of where in memory your data is stored so you can avoid causing errors which make your program crash with unhelpful error messages. (Oftentimes, an error message will tell you that the error occurred at a part your program that is nowhere near where it actually occurred as a result of the very same error that prompted the message.) Many instructions which are perfectly valid on their own can result in bizarre bugs when they bump into each other in memory.

For instance, let's say that the data registers are not sufficient to store your variables (possibly because your program requires more than eight variables), so you instead reserve in memory using the DS pseudo-OPcode. You can choose what units you are using to store this space with DS.B, DS.W, or DS.L (for Byte, Word, or Long), and the number of these units with an integer value. For instance, "DS.B 5" will store 5 bytes of space, while "DS.W 5" will store 5 words, or 10 bytes. However, if you reserve only 1 byte for some variable and later attempt to perform any kind of word- or long-sized operation on it, you will cause an error.

Far harder to fix are the kinds of bugs that can happen as a result of absolute addressing. For instance, if your program exists between $1000 and $2000 (the '$' means "hexadecimal") in the memory and your variables are stored between $2000 and $2100, then commands like "MOVE.W D3, $1100" or "ADDI.L #10, $2050" will overwrite parts of your program, including variables and instructions. Many times, this will not even raise any errors, but will simply lead to incorrect results.

Of course, this would be much easier to avoid if the Easy68k GUI had a pane indicating where your program starts and ends, but you can almost as easily execute the program, open the memory window from the simulation screen, jump to the location in memory you are writing to and ensure that it is not being used. Just do this every time you use absolute addressing and aren't sure the destination is available, and you probably won't have any issues of this particular kind!

(Alternately, you can simply never use absolute addressing as a destination register. Overwriting is much easier to avoid when using variables reserved with "DS"—however, this means you have to remember not to call commands that require more space than you've reserved, as explained before.)

Understanding the memory system becomes especially important when you are debugging your program, which is how you will spend most of your time in Easy68k (though to be fair, this is often true of programming in general—unless you're really good at it like me).

When testing your program, you have the option to run it all at once, which will execute the instructions in your program in the order the program counter points to them. (Barring branch instructions, this will be from the top down, starting at the first ORG pseudo-OPcode.) You can also run only one line at a time, make the simulator step through a few lines every second, or use some of the other simulation options.

Now that you understand how to use the basic 68k instructions in Easy68k and how to reduce your risk of memory-related issues, you're ready to start making your very own Easy68k programs! Just be sure to watch out for bugs. Not bugs in your program, mind you, but in Easy68k itself.

Part V: Bugs in Easy68k

As an assembly language environment, Easy68k tries to prioritize two things above all else: processing efficiency and as-direct-as-possible correspondence between human-readable instructions and machine code. Unfortunately, these two goals can sometimes conflict, resulting in bugs.

To clarify: these bugs are not mistakes, but intentional functionality which effectively constitutes a set of bugs in any context where we want consistent, direct, 68k-standard translations from OPcodes to hexadecimal machine code. For instance, the Motorola 68k manual states that:

- Assembled SUBA instructions are of the form 1001 + (register) + (OPmode) + (EA).
- Assembled SUBQ instructions are of the form 0101 + (data) + 1 + (size) + (EA).
- Never the twain shall meet.

However, if you try to assemble a SUBA instruction where the source is immediate data 8 or lower, you will find that the instruction is assembled as SUBQ instead, meaning the leading 1001 (9 in hexadecimal) will be replaced with a leading 0101 (5 in hexadecimal). This can be seen in the following example:

```
00000000                             1  *-----------------------------------------------------------
00000000                             2  * Title      : SUBA bug test
00000000                             3  * Written by : Anonymous
00000000                             4  * Date       : Unknown
00000000                             5  * Description: Showing off Shit68k's bullshit bugs
00000000                             6  *-----------------------------------------------------------
00000000                             7
00001000                             8  START   ORG    $1000               ; first instruction of program
00001000                             9
00001000                            10  * Put program code here
00001000                            11
00001000  98F8 3000                 12      SUBA.W      $3000,A4
00001004  98C4                      13      SUBA.W      D4,A4
00001006  98FC 0010                 14      SUBA.W      #16,A4
0000100A  5F8C                      15      SUBQ.L      #7,A4
0000100C  5F4C                      16      SUBA.W      #7,A4
0000100E  FFFF FFFF                 17      SIMHALT             ; halt simulator
00001012                            18
00001012                            19  * Put variables and constants here
00001012                            20
00001012                            21      END    START        ; last line of source

No errors detected
No warnings generated
```

Fig. 4: Look at this fucking horseshit.

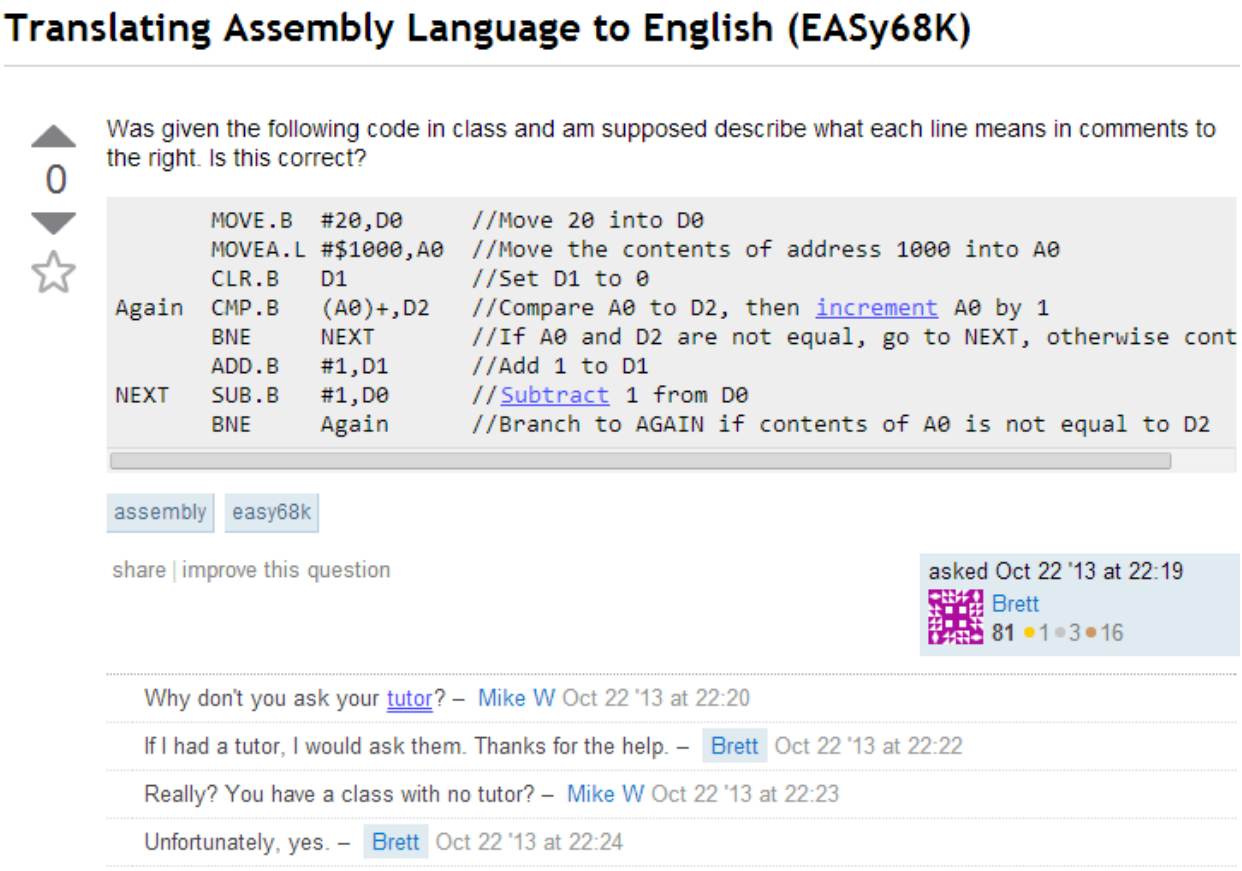
Again, this is done intentionally for efficiency's sake—if the SUBA instruction were assembled correctly, it would yield the machine code 98FC 0007, which you may notice is twice as long as 5F4C. However, assuming the user knows and cares about how 68k instructions in a program should affect the memory

(which is a prerequisite to using Easy68k in the first place), it is safe to assume that anyone using a SUBA command probably wants it to be parsed as such.

Now, you may be wondering when something like this might matter if the instruction has the same effect on its operands either way. Here is an example scenario:

Let's say you're building a disassembler in Easy68k. (This is a type program which reads hex machine code, built from assembled 68k instructions and stored somewhere in memory; translates it back into 68k assembly code; and then prints the parsed instructions in the console.) While on suicide watch, you discover the bug shown in Figure 4, which will ensure that you cannot correctly translate any SUBA instruction with immediate data 8 or lower. There is no workaround for this problem. It is impossible for you to tell the difference between machine code correctly translated from a SUBQ instruction and machine code incorrectly translated from a SUBA instruction. You will also encounter this problem for any other kind of instruction that has a special counterpart for immediate data. In other words, if your friends really cared about you, then they should have just you die.

If you happen to find any other Easy68k bugs, be sure to post them on the Internet someplace where people will hopefully find them. I recommend Stack Overflow, but don't bother posting bugs on the Easy68k forum, since it has been dead for a very, very long time. In fact, it seems to be a general rule that Stack Overflow has more information about Easy68k than the official documentation does.



Translating Assembly Language to English (EASy68K)

0

Was given the following code in class and am supposed describe what each line means in comments to the right. Is this correct?

```
        MOVE.B  #20,D0      //Move 20 into D0
        MOVEA.L #$1000,A0   //Move the contents of address 1000 into A0
        CLR.B   D1          //Set D1 to 0
Again   CMP.B   (A0)+,D2    //Compare A0 to D2, then increment A0 by 1
        BNE     NEXT        //If A0 and D2 are not equal, go to NEXT, otherwise cont
        ADD.B   #1,D1       //Add 1 to D1
NEXT    SUB.B   #1,D0       //Subtract 1 from D0
        BNE     Again       //Branch to AGAIN if contents of A0 is not equal to D2
```

assembly easy68k

share | improve this question

asked Oct 22 '13 at 22:19
Brett
81 ●1 ●3 ●16

Why don't you ask your tutor? – Mike W Oct 22 '13 at 22:20

If I had a tutor, I would ask them. Thanks for the help. – Brett Oct 22 '13 at 22:22

Really? You have a class with no tutor? – Mike W Oct 22 '13 at 22:23

Unfortunately, yes. – Brett Oct 22 '13 at 22:24

Fig. 5: Don't expect to learn about Easy68k in school, either. This is a battle you must fight alone.

Part VI: Coping with Easy68k: some final thoughts

If I still haven't dissuaded you from using Easy68k (presumably because you're being forced to use it for a class), I can at least tell you how to ease your pain through the sweet, sweet anesthetic of cutting corners. Here are some tips that your teacher most likely won't give you:

- Whenever possible, hardcode solutions rather than implementing them extensibly. Nuanced, generalizable structures like binary search trees, hashmaps, and arrays simply aren't feasible with Easy68k.
- Never, ever use the Easy68k GUI to write code. Instead, use a text editor designed for eukaryotes, like Sublime or Vim.
- Try writing a script in another language, like Python, in order to auto-generate a text file containing 68k code which you can then save as an .X68 file and run like any other Easy68k program. This is much, *much* faster than actually coding in Easy68k. However, don't just write the actual program in another language and give your teacher the auto-generated assembly files. You will be caught.

Since Easy68k is about as well-documented as it deserves to be, the knowledge contained in this guide should be more comprehensive and useful than you can expect to find anywhere else. However, there is still a lot about the emulator that not a man alive knows. Once you have experienced the nightmare firsthand and emerged with your sanity and GPA intact, feel free to add your personal insight to this guide and pass it along to others. Just make sure not to put your name on it.